CHARUSAT
CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

CE266: SOFTWARE ENGINEERING

Dec 2023 – April 2024

UNIT 5

Software Design

DEPSTAR

Content

- Design Concepts and Design Principle
- Architectural Design
- Component Level Design
- User Interface Design
- Web Application Design

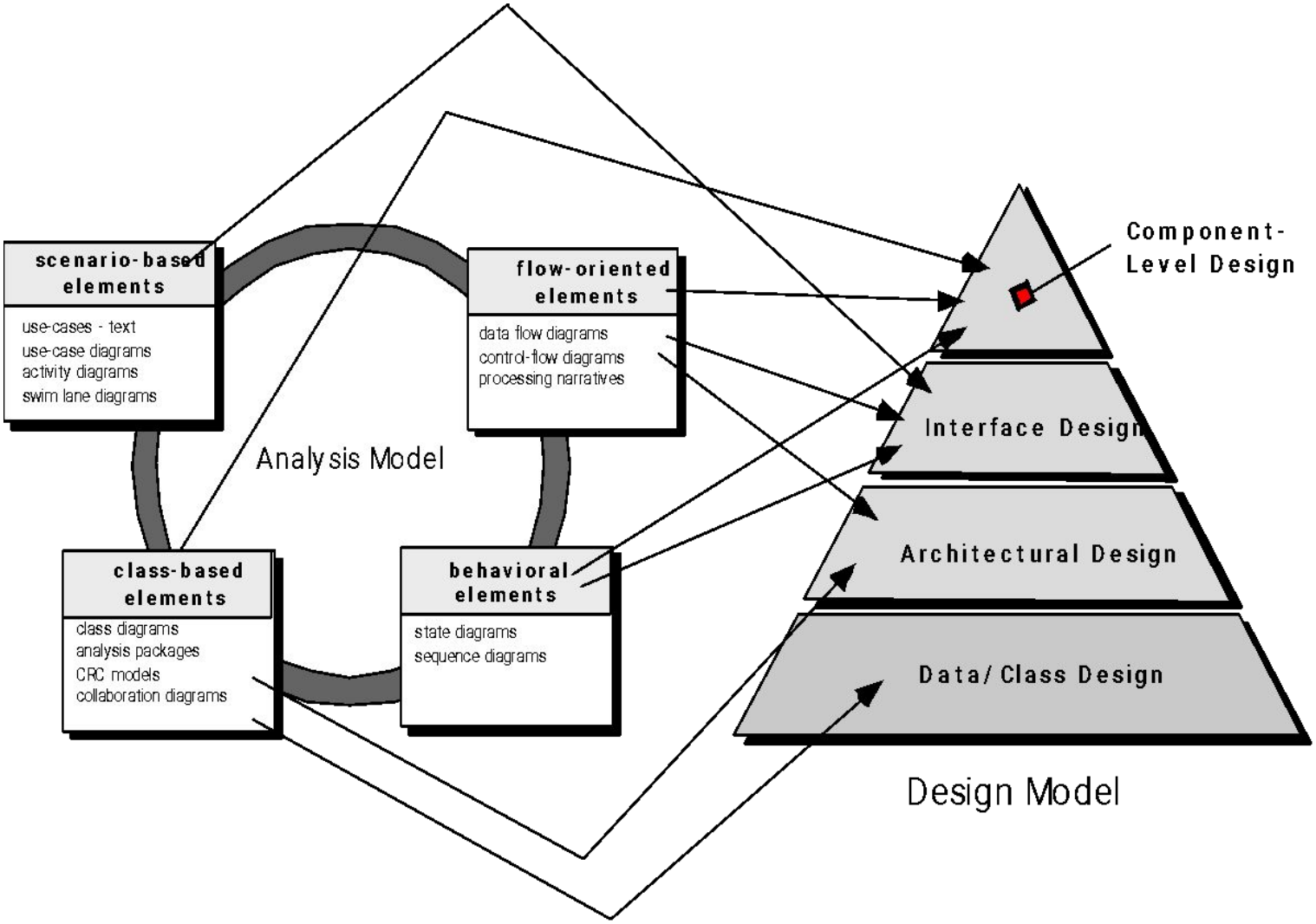
Design

- Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal.* He said:

  Good software design should exhibit:

  - *Firmness:* A program should not have any bugs that inhibit its function.
  - *Commodity:* A program should be suitable for the purposes for which it was intended.
  - *Delight:* The experience of using the program should be pleasurable one.

Analysis Model -> Design Model



Design and Quality

- the design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- the design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- the design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

- A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
  - For smaller systems, design can sometimes be developed linearly.
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

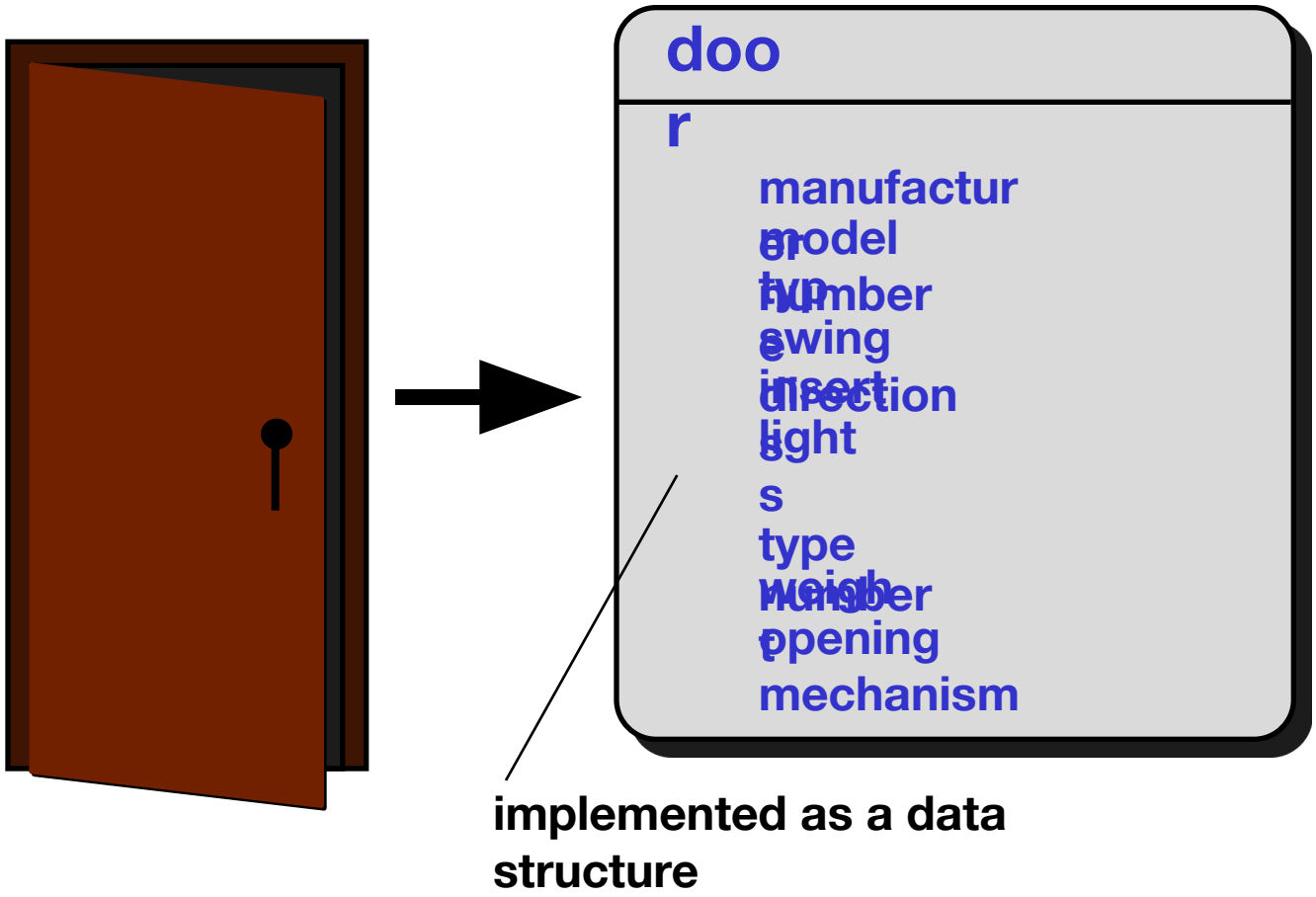
Design Principles

- The design process should not suffer from 'tunnel vision.'
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

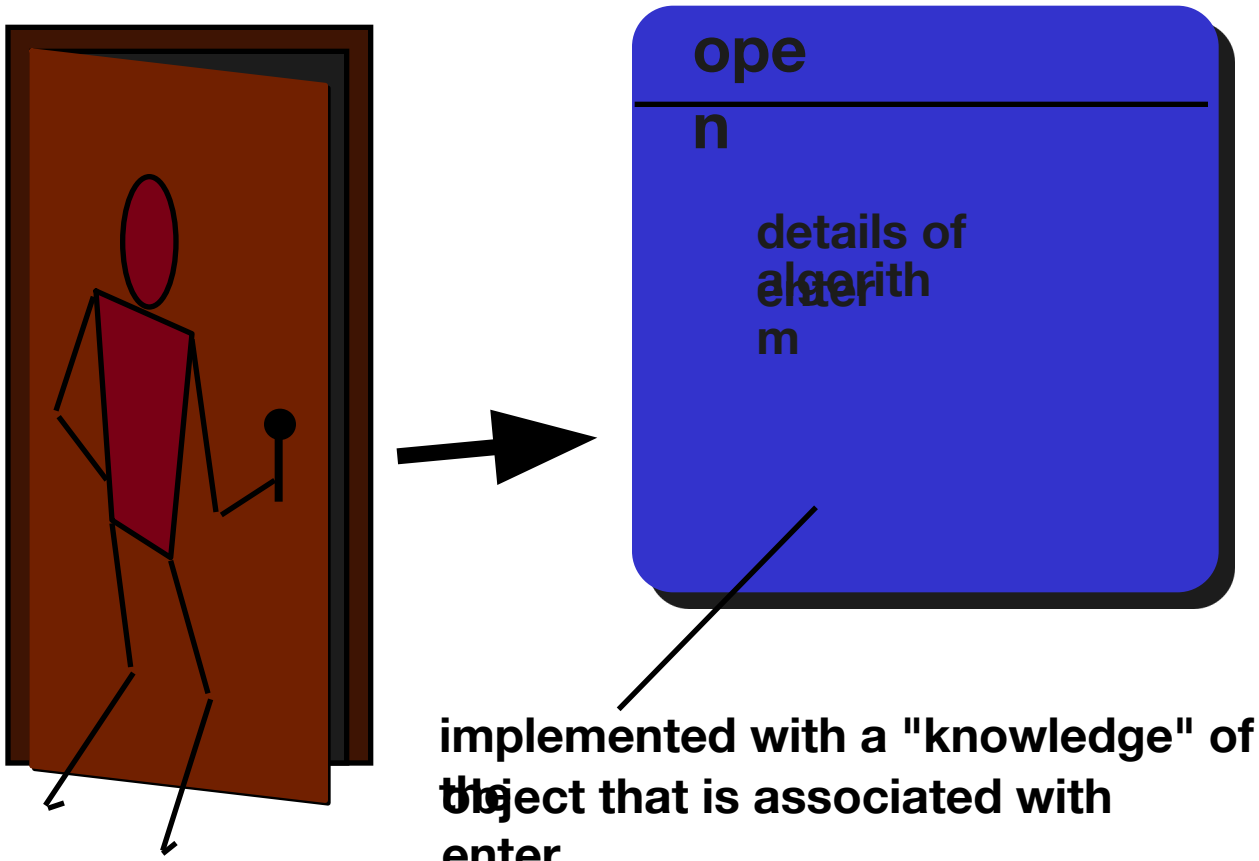
Fundamental Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concept
- Design Classes—provide design detail that will enable analysis classes to be implemented

Data Abstraction



Procedural Abstraction



Architecture

"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system." [SHA95a]

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name

Intent—describes the pattern and what it does

Motivation—provides an example of the problem

Applicability—notes specific design situations in which the pattern is applicable

Structure—describes the classes that are required to implement the pattern

Participants—describes the responsibilities of the classes that are required to implement the pattern

Collaborations—describes how the participants collaborate to carry out their responsibilities

Consequences—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

Related patterns—cross-references related design patterns

Separation of Concerns

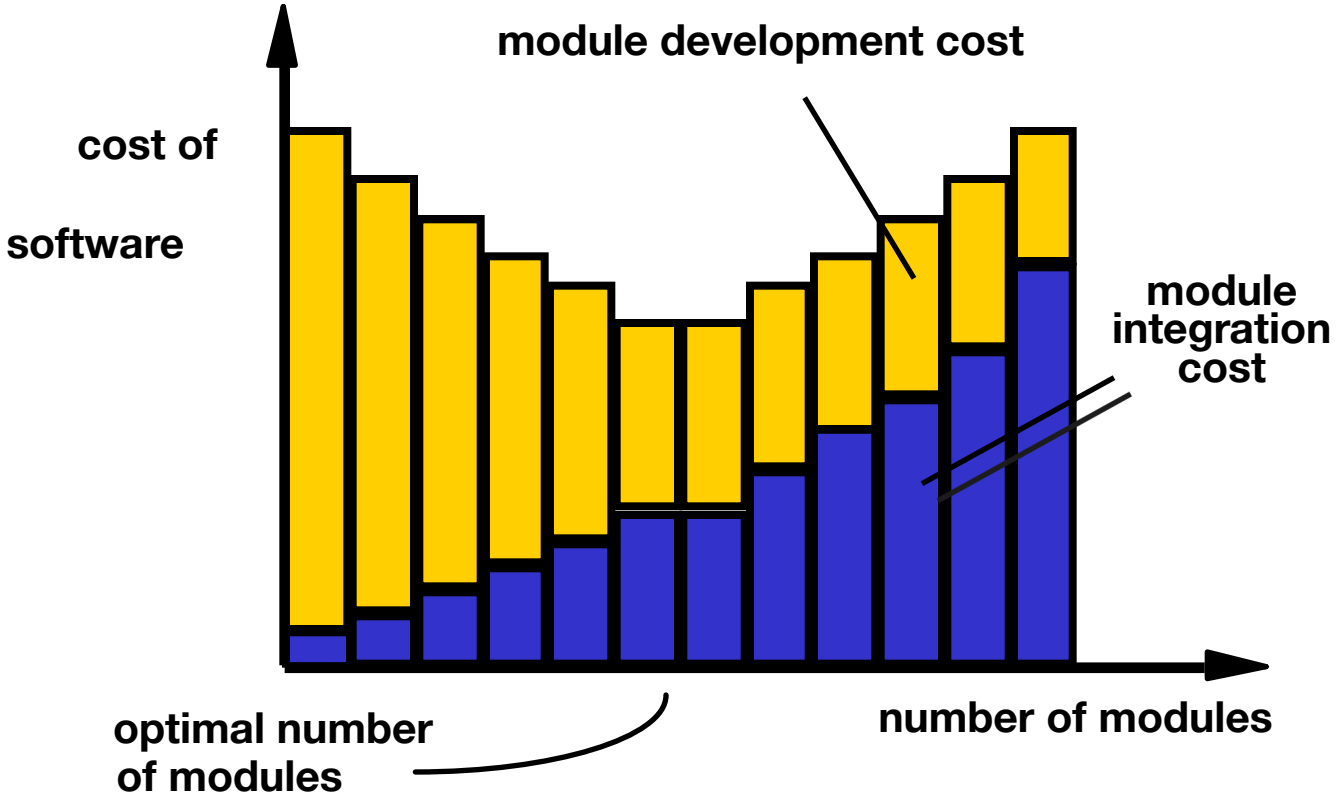
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

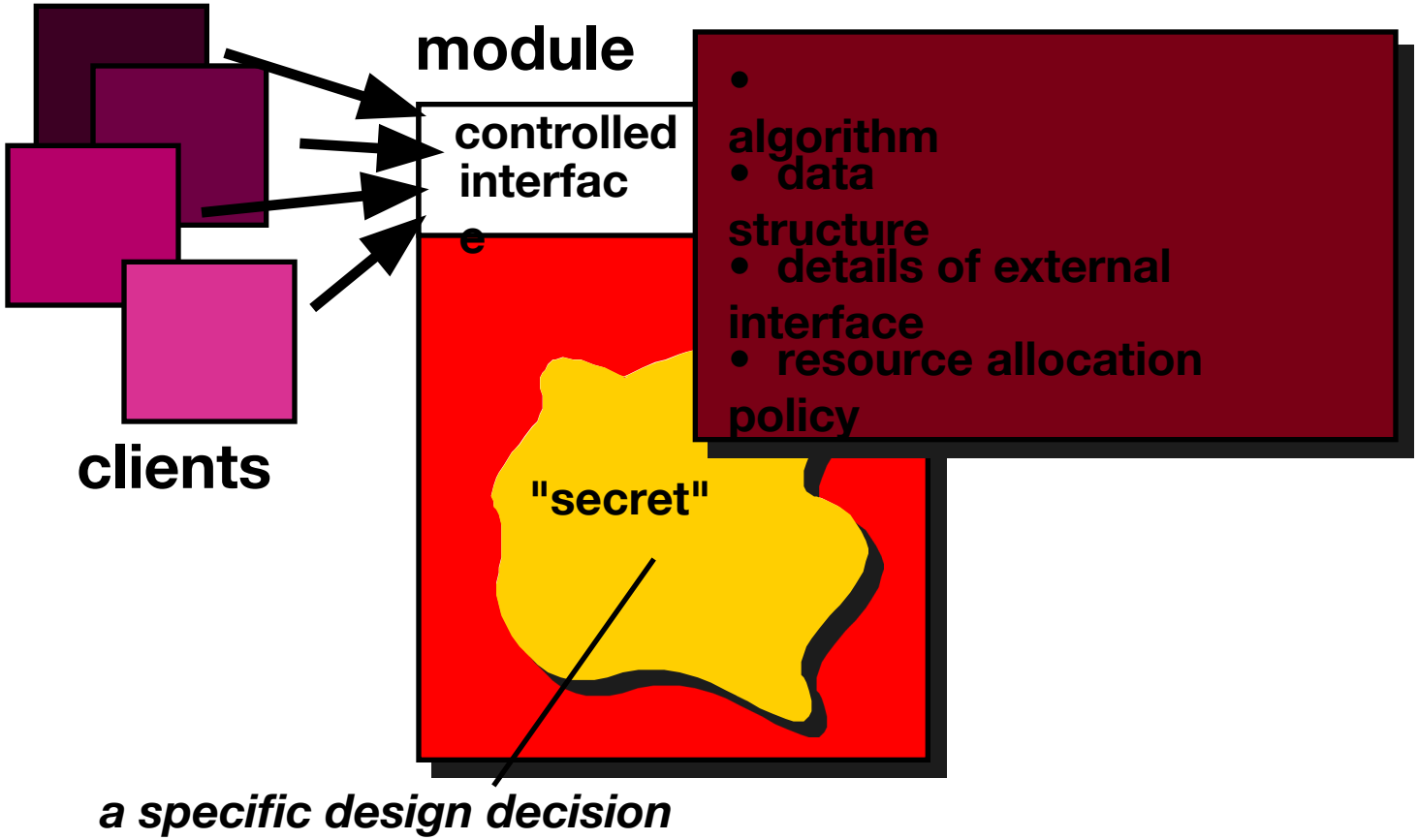
- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



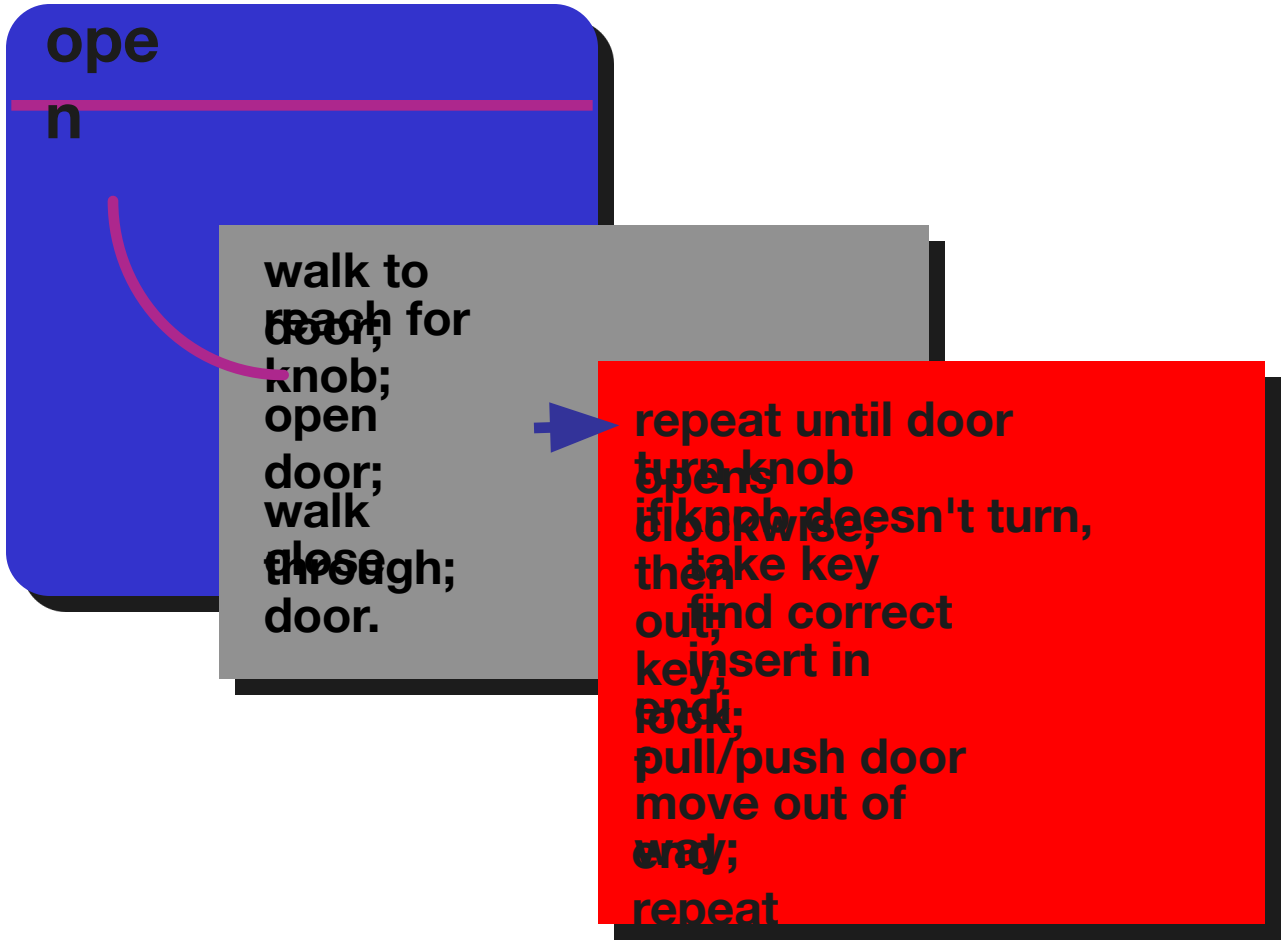
Information Hiding



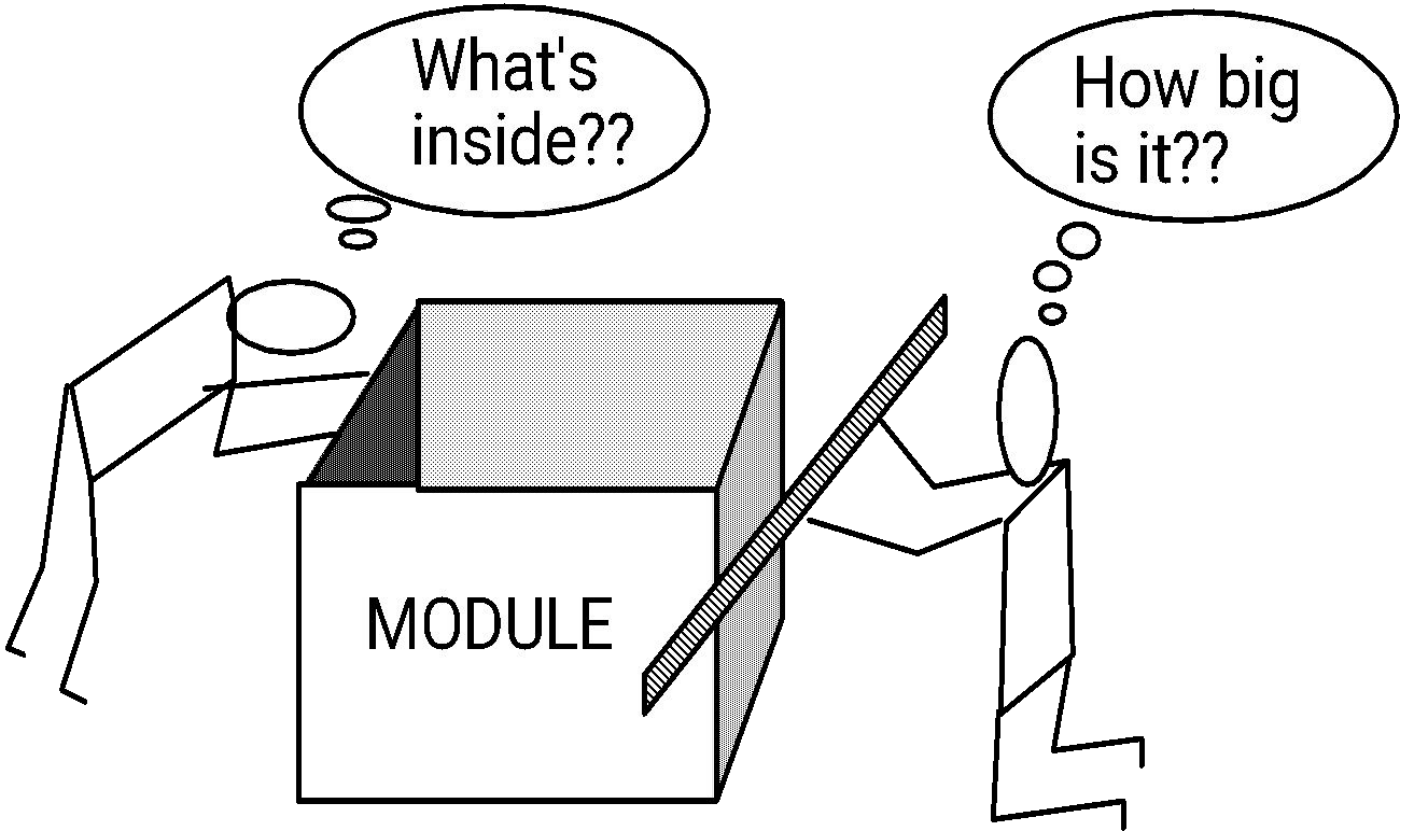
Why Information Hiding?

- reduces the likelihood of "side effects"
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Stepwise Refinement

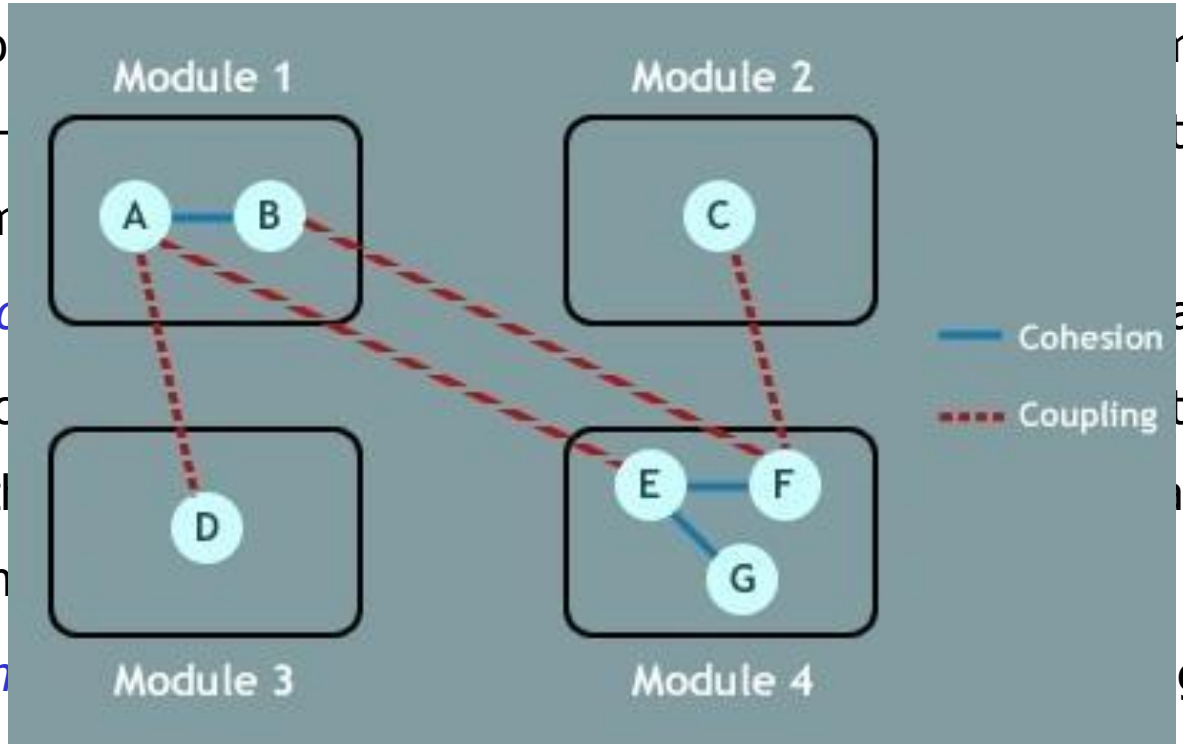


Sizing Modules: Two Views



Functional Independence

- Functio modules with "single- teraction with other m
- *Cohesi* a module.
  - A c tle interaction witl ted simply, a coh
- *Couplin* g modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.



Aspects

- Consider two requirements, *A* and *B.* Requirement *A* *crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the use-case **Access camera surveillance via the Internet.** A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and B* *cross-cuts* A*.
- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B**, of the requirement, *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp.

Refactoring

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
  - It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures

Architectural Design

- Why Architecture:
  - The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

Why is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together" [BAS03].

Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive System,* [IEE00]
  - to establish a conceptual framework and vocabulary for use during the design of software architecture,
  - to provide detailed guidelines for representing an architectural description, and
  - to encourage sound architectural design practices.
- The IEEE Standard defines an *architectural description* (AD) as a "a collection of products to document an architecture."
  - The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

Architectural Genres

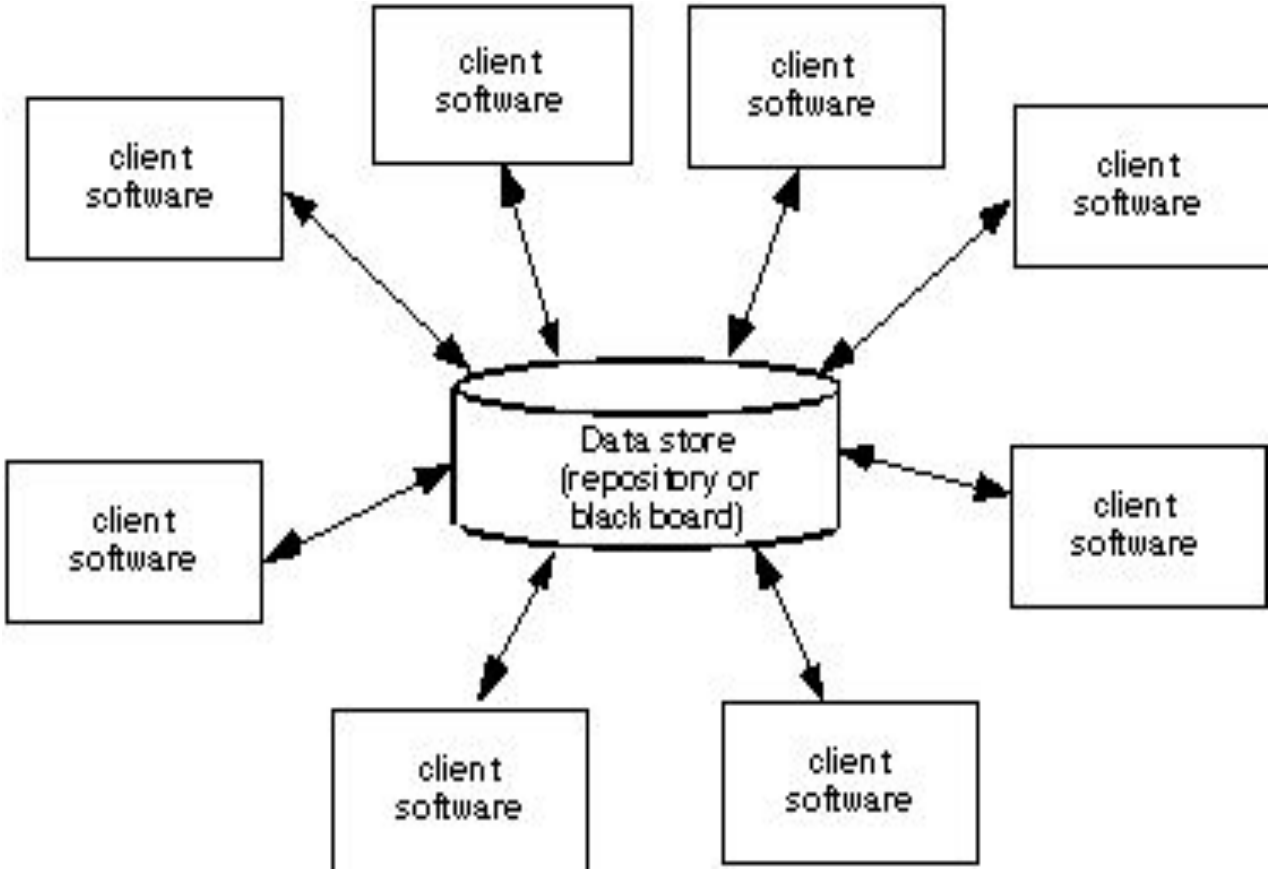
- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
  - For example, within the genre of *buildings*, you would encounter the following general styles: houses, apartment buildings, office buildings, industrial building, warehouses, and so on.
  - Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.

Architectural Styles

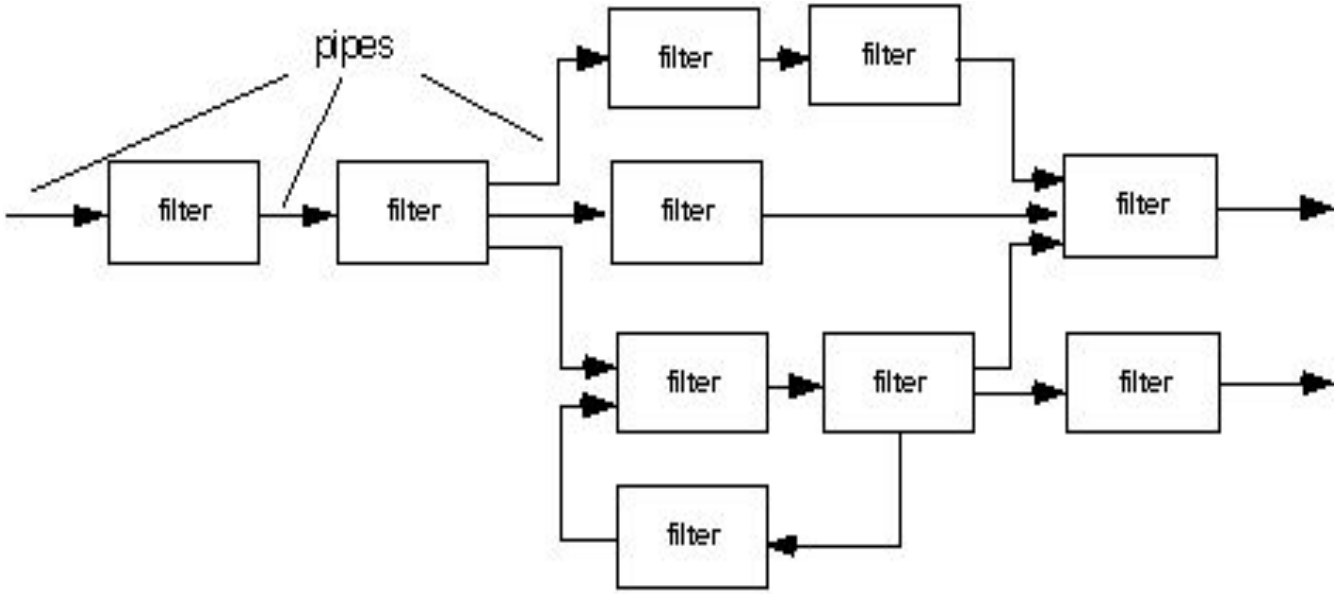
- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

Each style describes a system category that encompasses: (1) a set of components (e.g., a database, computational modules) that perform a function required by a system, (2) a set of connectors that enable "communication, coordination and cooperation" among components, (3) constraints that define how components can be integrated to form the system, and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

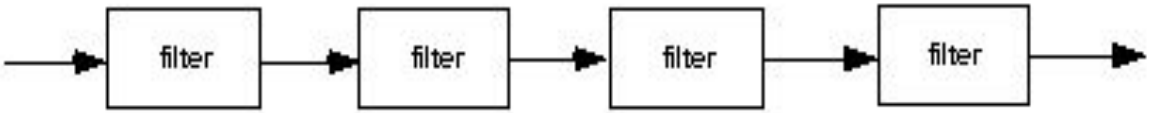
Data-Centered Architecture



Data Flow Architecture



(a) pipes and filters

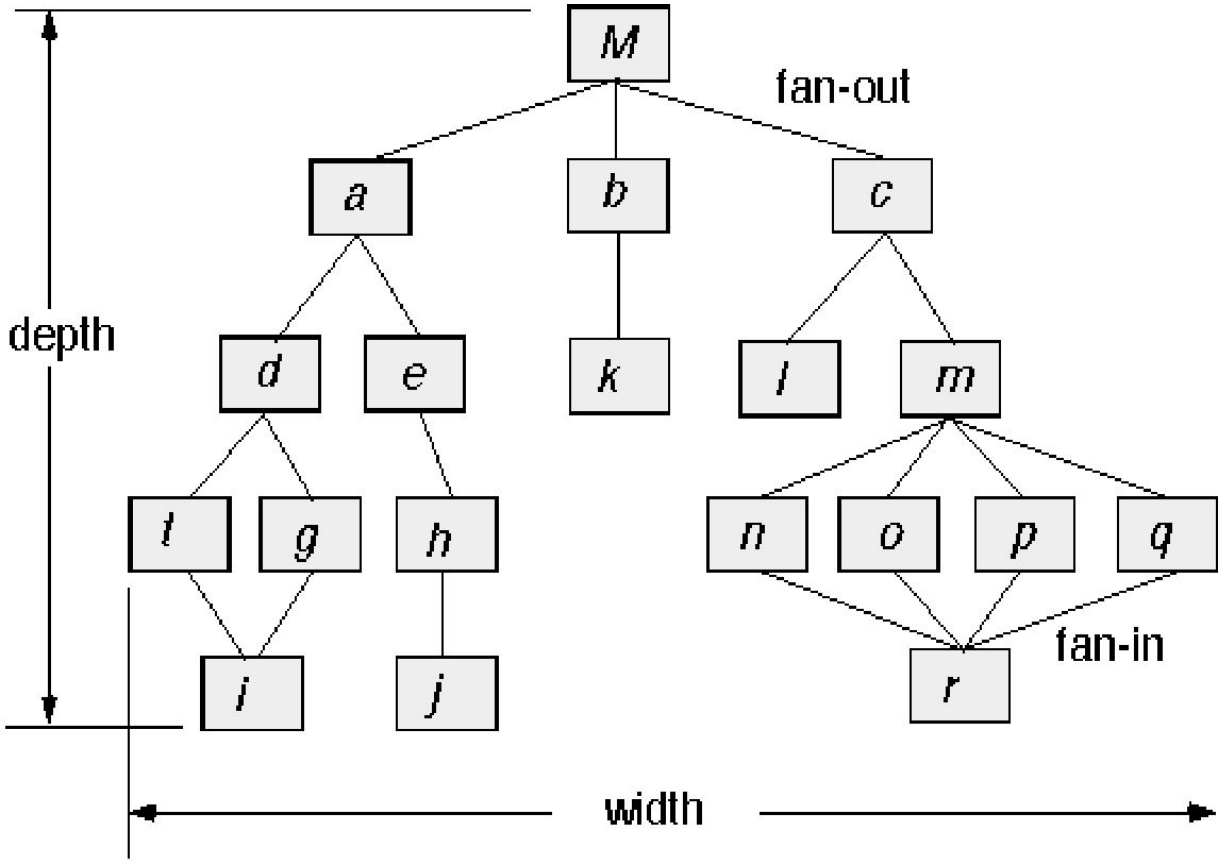


(b) batch sequential

Contd…

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A pipe and filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- However, the filter does not require knowledge of the working of its neighboring filters.

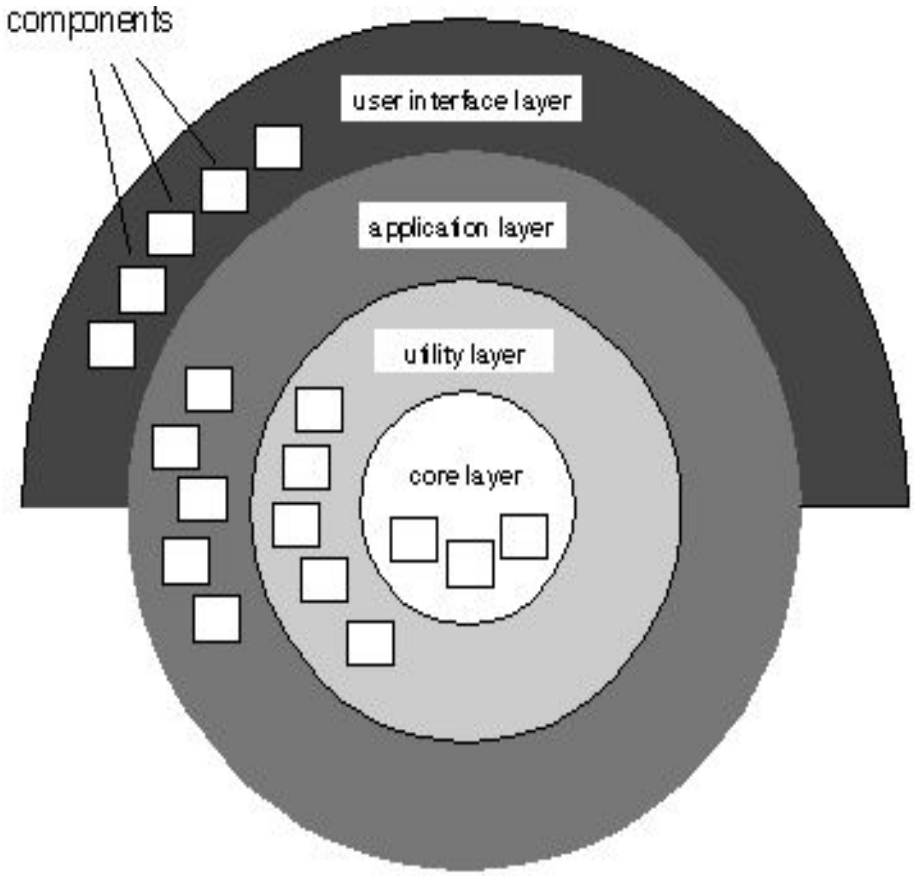
Call and Return Architecture



Contd…

- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.
- A number of sub styles exist within this category:
  - Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.
  - Remote procedure call architectures. The components of a main program/ subprogram architecture are distributed across multiple computers on a network.

Layered Architecture



Contd…

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.

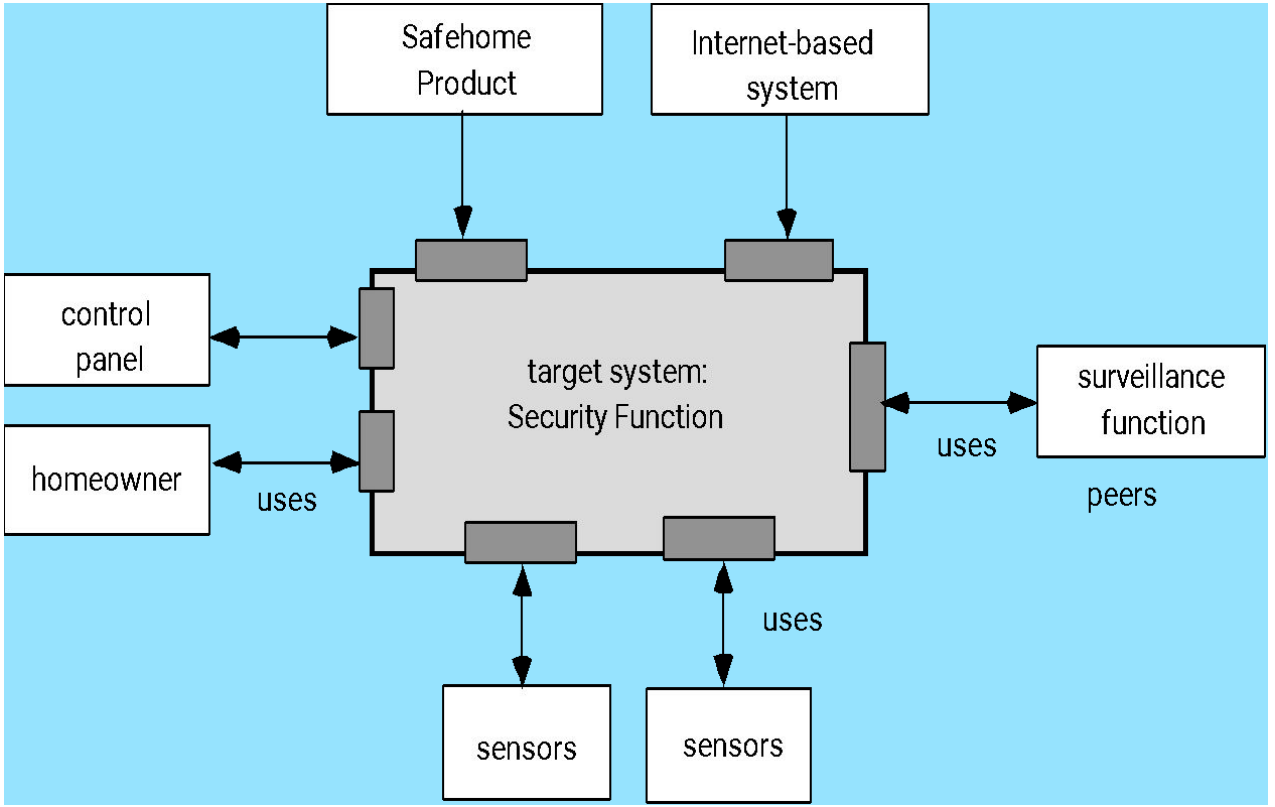
Architectural Patterns

- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
  - *operating system process management* pattern
  - *task scheduler* pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
  - an *application level persistence* pattern that builds persistence features into the application architecture
- Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment
  - A *broker* acts as a 'middle-man' between the client component and a server component.

Architectural Design

- The software must be placed into context
  - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
  - An *archetype* is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software components that implement each archetype

Architectural Context



Archetypes

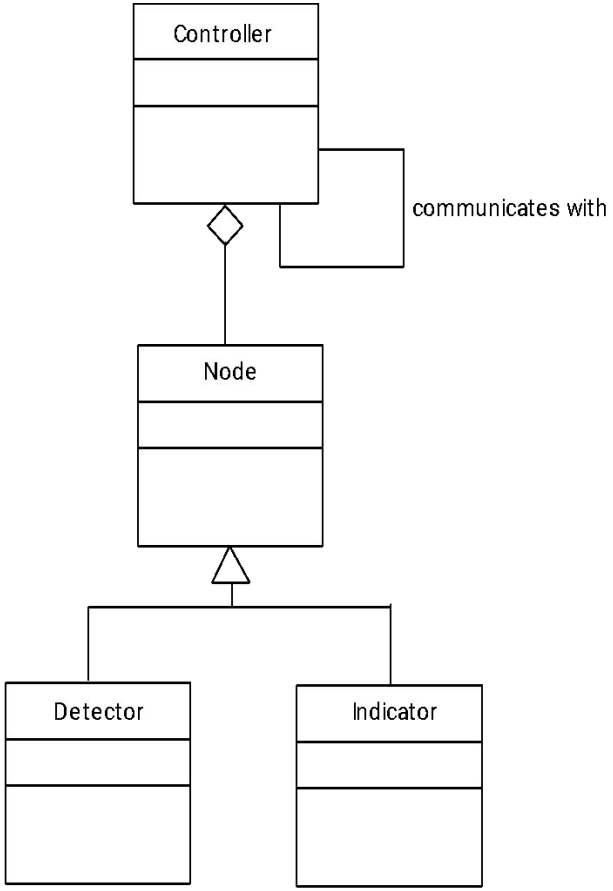


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

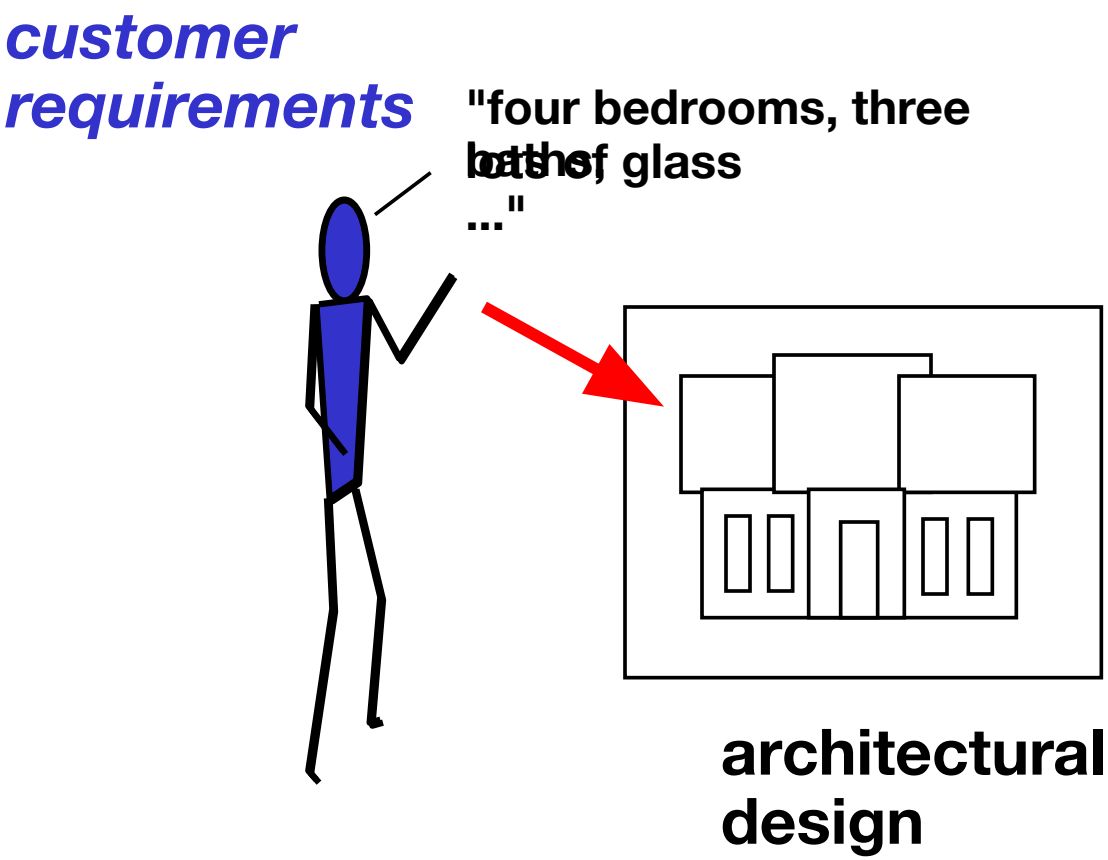
Analyzing Architectural Design

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
    - module view
    - process view
    - data flow view
4. Evaluate quality attributes by considered each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

ADL

- *Architectural description language* (ADL) provides a semantics and syntax for describing a software architecture
- Provide the designer with the ability to:
  - decompose architectural components
  - compose individual components into larger architectural blocks and
  - represent interfaces (connection mechanisms) between components.

An Architectural Design Method



Data Design at the Architectural Level

- The challenge is extract useful information from the data environment, particularly when the information desired is cross-functional.
- To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in databases (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information.
- However, the existence of multiple databases, their different structures, and the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment.
- An alternative solution, called a data warehouse, adds on additional layer to the data architecture.
- A data warehouse is a separate data environment that is not directly integrated with day-to-day applications that encompasses all data used by a business.

Data Design at the Component Level

- At the component level, data design focuses on specific data structures required to realize the data objects to be manipulated by a component.
  - Refine data objects and develop a set of data abstractions
  - Implement data object attributes as one or more data structures
  - Review data structures to ensure that appropriate relationships have been established

Contd…

- **Set of principles for data specification**:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low level data design decisions should be deferred until late in the design process.
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

Component Level Design

- What is a component?
- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
  - "… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.""
- *OO view:* a component contains a set of collaborating classes
- *Conventional view:* a component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

- Component-level design, also called procedural design, occurs after data, architectural, and interface designs have been established.
- Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.
- The intent is to translate the design model into operational software.
- But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low.
- Component a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

Basic Design Principles

- The Open-Closed Principle (OCP). *"A module [component] should be open for extension but closed for modification.*
- The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes.*
- Dependency Inversion Principle (DIP). *"Depend on abstractions. Do not depend on concretions."*
- The Interface Segregation Principle (ISP). *"Many client-specific interfaces are better than one general purpose interface.*
- The Release Reuse Equivalency Principle (REP). *"The granule of reuse is the granule of release."*
- The Common Closure Principle (CCP). *"Classes that change together belong together."*
- The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together."*

Design Guidelines

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- Interfaces
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC)
- Dependencies and Inheritance
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Function Oriented Approach

- The following are the salient features of a typical function-oriented design approach:

1. A system is viewed as something that performs a set of functions. Starting at this highlevel view of the system, each function is successively refined into more detailed functions.

For example, consider a function create-new-library member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This function may consist of the following sub-functions:

- assign-membership-number
- create-member-record
- print-bill

Each of these sub-functions may be split into more detailed sub-functions and so on.

Contd…

2. The system state is centralized and shared among different functions, e.g. data such as member- records is available for reference and updating to several functions such as:

- create-new-member
- delete-member
- update-member-record

Object Oriented Approach

- In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information.
- For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects.
- Objects have their own internal data which define their state. Similar objects constitute a class.
- In other words, each object is a member of some class. Classes may inherit features from super class.
- Conceptually, objects communicate by message passing.

Function-Oriented Vs. Object-Oriented Design

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real world functions such as sort, display, track, etc., but real-world entities such as employee, picture, machine, radar system, etc.
- For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee record, get-employee-address, etc. but by designing objects such as employees, departments, etc.
- In object-oriented design, software is not developed by designing functions such as update-employee- record, get-employee-address, etc., but by designing objects such as employee, department, etc.
- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system.

Contd…

- For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system.
- Objects communicate by passing messages. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or the other the real-world functions must be implemented.
- Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

Cohesion

- Conventional view:
  - the "single-mindedness" of a module
- OO view:
  - *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- Levels of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - utility

Contd…

- Cohesion is an indication of the relative functional strength of a module.
- A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- Cohesion is a measure of functional strength of a module.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- By the term functional independence, we mean that a cohesive module performs a single task or function.

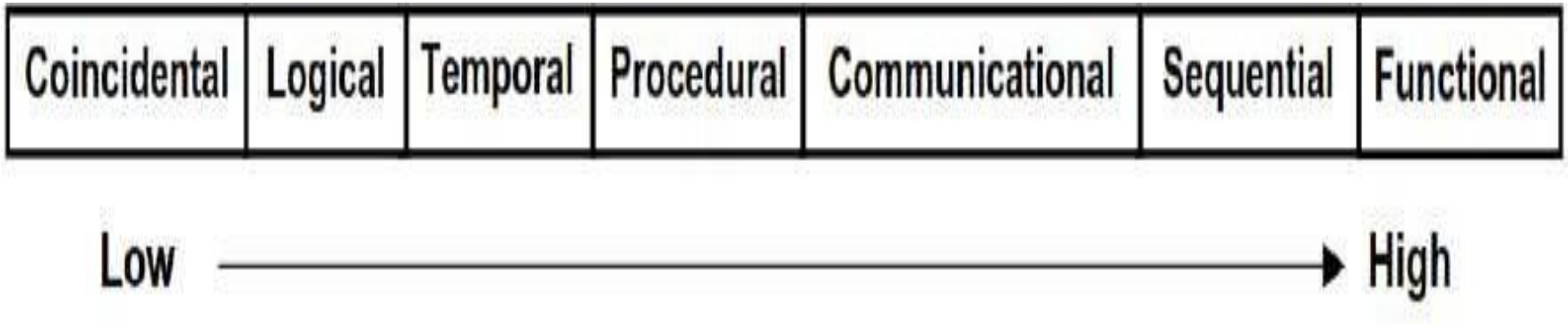
Coupling

- Conventional view:
  - The degree to which a component is connected to other components and to the external world
- OO view:
  - a qualitative measure of the degree to which classes are connected to one another
- Level of coupling
  - Content
  - Common
  - Control
  - Stamp
  - Data

Contd…

- Coupling is an indication of the relative interdependence among modules.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- If two modules interchange large amounts of data, then they are highly interdependent.
- The degree of coupling between two modules depends on their interface complexity.

Classification of Cohesion



Coincidental cohesion

- A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.
- In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.
- For example, in a transaction processing system (TPS), the get-input, print-error, and summarize- members functions are grouped into one module.

Logical Cohesion

- A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc.
- An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

Temporal Cohesion

- When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion.
- The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Procedural Cohesion

- A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message

Communicational Cohesion

- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or astack.

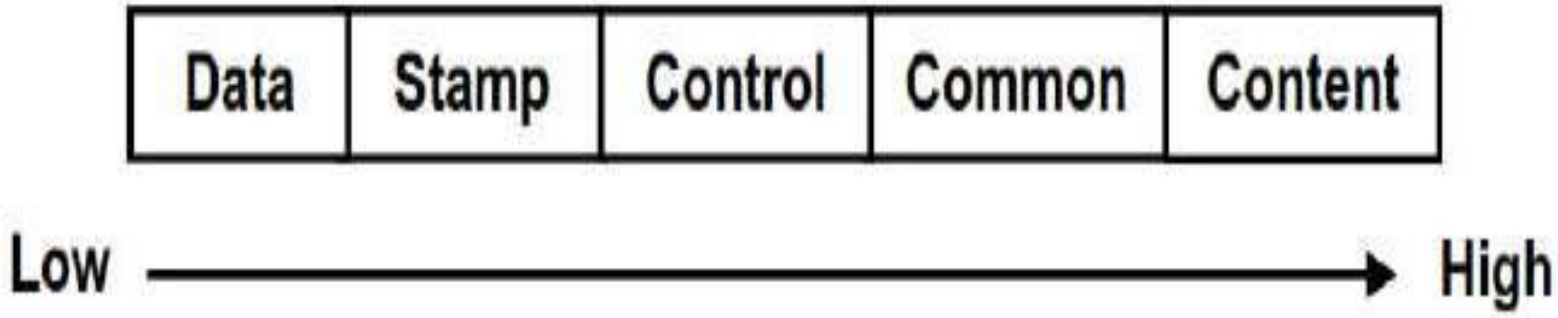
Sequential Cohesion

- A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.
- For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

Functional Cohesion

- Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion.
- Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.
- Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion.
- Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

Classification of Coupling



Data Coupling

- Two modules are data coupled, if they communicate through a parameter.
- An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc.
- This data item should be problem related and not used for the control purpose.

Stamp coupling

- Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

- **Control coupling**

  Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module.

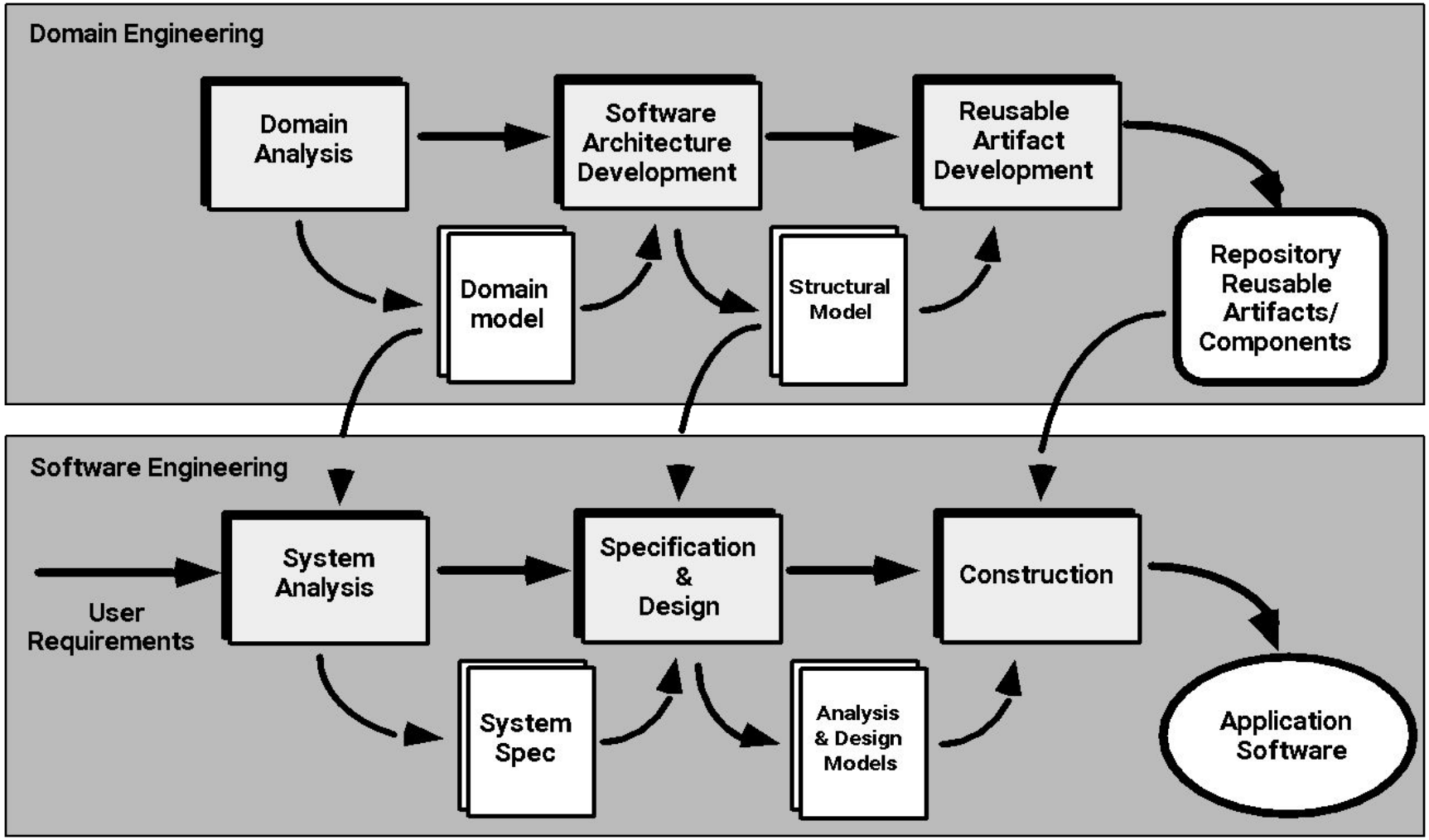
- **Common coupling**

  Two modules are common coupled, if they share data through some global data items.

- **Content coupling**

  Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

The CBSE Process



CBSE Activities

- Component qualification
- Component adaptation
- Component composition
- Component update

Qualification

Before a component can be used, you must consider:

- application programming interface (API)
- development and integration tools required by the component
- run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
- service requirements including operating system interfaces and support from other components
- security features including access controls and authentication protocol
- embedded design assumptions including the use of specific numerical or non-numerical algorithms
- exception handling

Adaptation

The implication of "easy integration" is:

(1) that consistent methods of resource management have been implemented for all components in the library;

(2) that common activities such as data management exist for all components, and

(3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

Composition

- An infrastructure must be established to bind components together
- Architectural ingredients for composition include:
  - Data exchange model
  - Automation
  - Structured storage
  - Underlying object model

User Interface Design

- *User interface design* creates an effective communication medium between a human and a computer.
- Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Design Rules for User Interface

1. Place the user in control.
2. Reduce the users' memory load.
3. Make the interface consistent.

Place the user in control

1. **Define interaction modes in a way that does not force a user into unnecessary or undesired actions.**

- An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way.

2. **Provide for flexible interaction.**

- Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.

3. Allow user interaction to be interruptible and undoable.

- Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else.

4. Streamline interaction as skill levels advance and allow the interaction to be customized.

- Users often find that they perform the same sequence of interactions repeatedly.

5. Hide technical internals from the casual user.

- The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

6. Design for direct interaction with objects that appear on the screen.

- The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

Reduce the User's Memory Load

1. **Reduce demand on short-term memory.**

- When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

2. **Establish meaningful defaults**.

- The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

3. **Define shortcuts that are intuitive**.

- When mnemonics are used to accomplish a system function, the mnemonic should be tied to the action in a way that is easy to remember.

4. **The visual layout of the interface should be based on a real-world metaphor**.

- This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

5. **Disclose information in a progressive fashion.**

- The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction.

Make the interface consistent

1. **Allow the user to put the current task into a meaningful context.**

- Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators that enable the user to know the context of the work at hand.

2. **Maintain consistency across a family of applications.**

- A set of applications should all implement the same design rules so that consistency is maintained for all interaction.

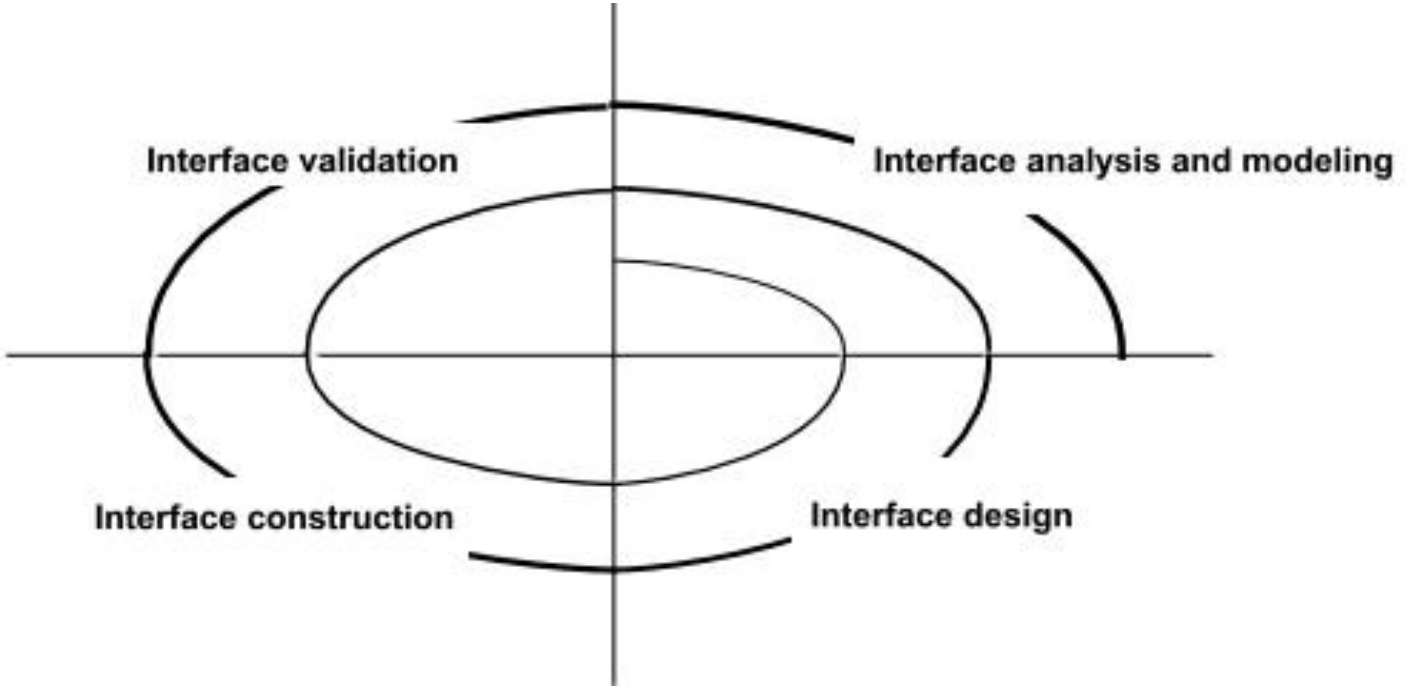
3. **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.**

- Once a particular interactive sequence has become a de facto standard, the user expects this in every application he encounters.

User Interface Design Models

- User model — a profile of all end users of the system
- Design model — a design realization of the user model
- Mental model (system perception) — the user's mental image of what the interface is
- Implementation model — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics

User Interface Design Process



Interface Analysis

- Interface analysis means understanding
  - (1) the people (end-users) who will interact with the system through the interface;
  - (2) the tasks that end-users must perform to do their work,
  - (3) the content that is presented as part of the interface
  - (4) the environment in which these tasks will be conducted.

User Analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology the sits behind the interface?

Task Analysis and Modeling

- Answers the following questions …
  - What work will the user perform in specific circumstances?
  - What tasks and subtasks will be performed as the user does the work?
  - What specific problem domain objects will the user manipulate as work is performed?
  - What is the sequence of work tasks—the workflow?
  - What is the hierarchy of tasks?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved

Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

WebApp Interface Design

- *Where am I?* The interface should
  - provide an indication of the WebApp that has been accessed
  - inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
  - what functions are available?
  - what links are live?
  - what content is relevant?
- *Where have I been, where am I going?* The interface must facilitate navigation.
  - Provide a "map" (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

Design & WebApps

- *When should we emphasize WebApp design?*
  - when content and function are complex
  - when the size of the WebApp encompasses hundreds of content objects, functions, and analysis classes
  - when the success of the WebApp will have a direct impact on the success of the business

Design & WebApp Quality

- Security
  - Rebuff external attacks
  - Exclude unauthorized access
  - Ensure the privacy of users/customers
- Availability
  - the measure of the percentage of time that a WebApp is available for use
- Scalability
  - **Can** the WebApp and the systems with which it is interfaced handle significant variation in user or transaction volume
- Time to Market

Quality Dimensions for End-Users

- *Time*
  - How much has a Web site changed since the last upgrade?
  - How do you highlight the parts that have changed?
- *Structural*
  - How well do all of the parts of the Web site hold together.
  - Are all links inside and outside the Web site working?
  - Do all of the images work?
  - Are there parts of the Web site that are not connected?
- *Content*
  - Does the content of critical pages match what is supposed to be there?
  - Do key phrases exist continually in highly-changeable pages?
  - Do critical pages maintain quality content from version to version?
  - What about dynamically generated HTML pages?

Quality Dimensions for End-Users

- ***Accuracy and Consistency***
  - Are today's copies of the pages downloaded the same as yesterday's? Close enough?
  - Is the data presented accurate enough? How do you know?
- ***Response Time and Latency***
  - Does the Web site server respond to a browser request within certain parameters?
  - In an E-commerce context, how is the end to end response time after a SUBMIT?
  - Are there parts of a site that are so slow the user declines to continue working on it?
- ***Performance***
  - Is the Browser-Web-Web site-Web-Browser connection quick enough?
  - How does the performance vary by time of day, by load and usage?
  - Is performance adequate for E-commerce applications?

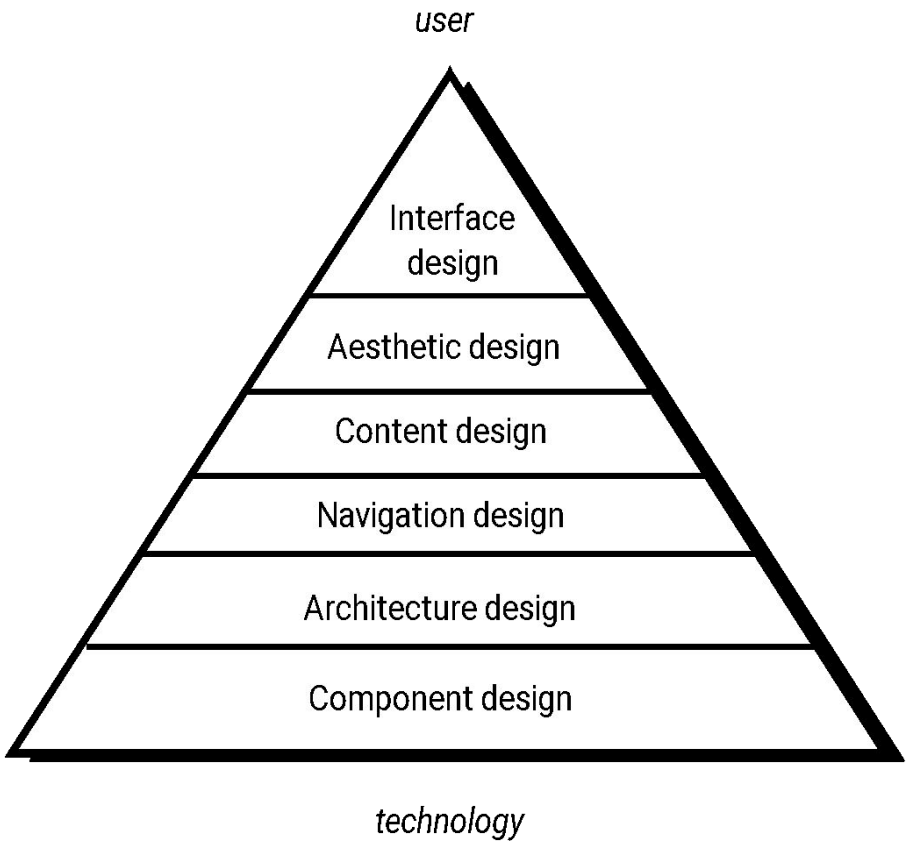
WebApp Design Goals

- Consistency
  - Content should be constructed consistently
  - Graphic design (aesthetics) should present a consistent look across all parts of the WebApp
  - Architectural design should establish templates that lead to a consistent hypermedia structure
  - Interface design should define consistent modes of interaction, navigation and content display
  - Navigation mechanisms should be used consistently across all WebApp elements

WebApp Design Goals

- Identity
  - Establish an "identity" that is appropriate for the business purpose
- Robustness
  - The user expects robust content and functions that are relevant to the user's needs
- Navigability
  - designed in a manner that is intuitive and predictable
- Visual appeal
  - the look and feel of content, interface layout, color coordination, the balance of text, graphics and other media, navigation mechanisms must appeal to end-users
- Compatibility
  - With all appropriate environments and configurations

WebE Design Pyramid



Interface Design Principles

- Anticipation—A WebApp should be designed so that it anticipates the use's next move.
- Communication—The interface should communicate the status of any activity initiated by the user
- Consistency—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- Controlled autonomy—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- Efficiency—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.

Contd…

- Focus—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- Fitt's Law—"The time to acquire a target is a function of the distance to and size of the target."
- Human interface objects—A vast library of reusable human interface objects has been developed for WebApps.
- Latency reduction—The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- Learnability— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

Contd…

- Maintain work product integrity—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- Readability—All information presented through the interface should be readable by young and old.
- Track state—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- Visible navigation—A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them."

Aesthetic Design

- Don’t be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don’t extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

Content Design

- Develops a design representation for content objects
  - For WebApps, a content object is more closely aligned with a data object for conventional software
- Represents the mechanisms required to instantiate their relationships to one another.
  - analogous to the relationship between analysis classes and design components
- A content object has attributes that include content-specific information and implementation-specific attributes that are specified as part of design

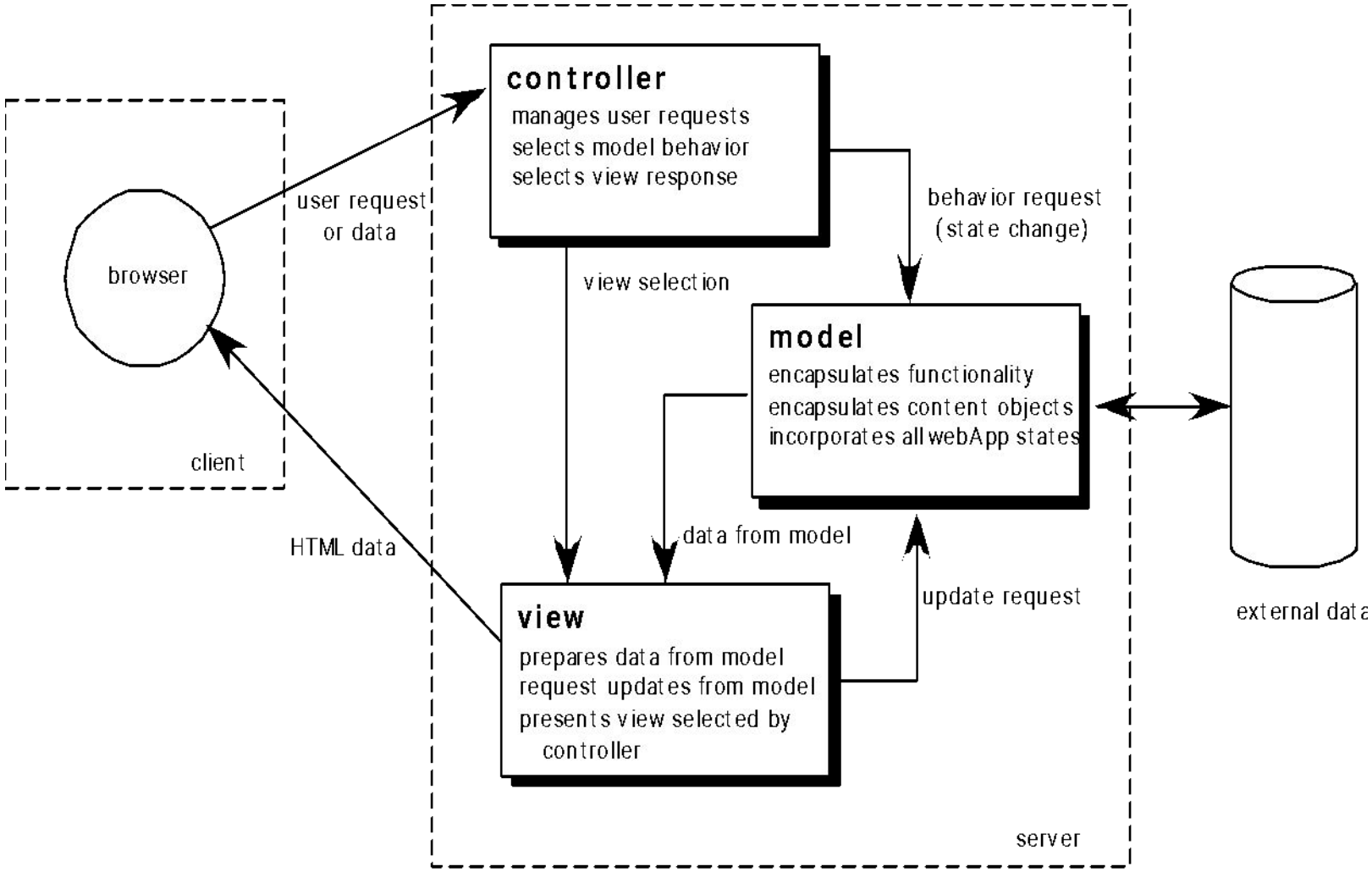
Architecture Design

- *Content architecture* focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation.
  - The term information architecture is also used to connote structures that lead to better organization, labeling, navigation, and searching of content objects.
- *WebApp architecture* addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.
- Architecture design is conducted in parallel with interface design, aesthetic design and content design.

MVC Architecture

- The *model* contains all application specific content and processing logic, including
  - all content objects
  - access to external data/information sources,
  - all processing functionality that are application specific
- The *view* contains all interface specific functions and enables
  - the presentation of content and processing logic
  - access to external data/information sources,
  - all processing functionality required by the end-user.
- The *controller* manages access to the model and the view and coordinates the flow of data between them.

MVC Architecture



Navigation Design

- Begins with a consideration of the user hierarchy and related use-cases
  - Each actor may use the WebApp somewhat differently and therefore have different navigation requirements
- As each user interacts with the WebApp, she encounters a series of *navigation semantic units* (NSUs)
  - NSU—"a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements"

Component-Level Design

- WebApp components implement the following functionality
  - perform localized processing to generate content and navigation capability in a dynamic fashion
  - provide computation or data processing capability that are appropriate for the WebApp's business domain
  - provide sophisticated database query and access
  - establish data interfaces with external corporate systems.


Thank you!